ELITE

MACHINE CODE

Made Easy

Do you remember the good old days when your new computer was still shiny and just out of its box? A few compulsory games of *Space Invaders* later and you start to write your first fruit machine game in the old standby Basic. After the not-too-wondrous results of this experiment, its on to the Pacmen for you. A couple of days later, you find yourself hacking about on the machine at a time miles past even your bedtime. It would seem you've become a programmer.

But weren't the results disappointing? If your hobby (or indeed even your profession) has started to pall, what you need is a good dose of machine code.

Add sparkle to your arcade action games, make your lasers zap aliens that no laser has zapped before, and utilise your utilities to their full potential. Make your aliens glide smoothly across your screen and make even the Spectrum appear to have *real* sound. Extend your machine's Basic to do all the things you could ever want it to do. Oh, the joys of machine code!

'Ah, but. . .', we hear you say. 'Isn't machine code that difficult subject I read about in my manual?'. Well, yes it is, but it isn't really all *that* hard, and with this series of cards, anyone will be able to get to grips with the subject.

Every popular home computer except the Dragon is covered fully in these cards, and very soon you'll find yourself writing programs to rival anything you can buy. Well, fairly soon. . . . .

So if you own or are about to buy a home computer, the chances are that it will be built around either a Z80 processor chip (e.g. Sinclair, Sharp and Memotech) or the 6502 chip (Commodore, Atari and Acorn). The only well-known home micro not based on one of these two chips is the Dragon, which uses a 6809.

Both these processors, and therefore the machines containing them, conform to particular standards, in that they are known as 8-bit chips, and the basic operation of each is very similar.

At this point you may well ask, what is an 8-bit chip? Bit stands for 'binary digit', that is, a piece of silicon that either conforms to 0 (off) or 1 (on) at any particular time, and this bit is the basis for all information transferal and manipulation.

By being described as 8-bit, then, these machines operate on eight of these bits at a time. Jargon being so prevalent in computing, this group of eight bits has become known as a 'byte'. (Just for the record, a group of four bits is called a nibble!) As each of the bits in a byte can be either off or on (represented by 0 or 1), it follows that a byte can be seen as eight adjacent zeros or ones. A quick mathematical deduction shows that if all the possible combinations of zeros and ones from eight 0's to 8 1's are added up, there are 256 different values that a byte can hold. It just so happens that 256 is also 2 to the power of 8, so there is nothing to stop us using a byte to represent a number.

Remember that 2^1 is 2, 2^2 (i.e. 2 squared) is 4, 2^3 (2 cubed) is 8, etc. If we represent a byte as shown below and make the rightmost bit (bit 0) represent 2^0 (which equals 1) and each successive bit up to bit 7 represent 2 to the power of that bit, then the byte can be seen to be able to represent any value from 0 to 255. For instance, if all the bits were 1, then the value of the byte would be $(2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7)$, which equals 1+2+4+8+16+32+64+128), and of course this adds up to 255.

As we have already said, there are 256 possible values for an 8-bit byte, and we have also discovered that the highest value for such a byte is 255, and that 256 is 2^8. Therefore, it follows that the highest number can be held in n bits is $(2^7) - 1$. The latest development in the personal computer world - the 16-bit micros - can, therefore, handle numbers up to (2^{16})-1, which equals 65535.

Owners of normal 8-bit machines will probably have heard of this number before - it is in fact the highest possible address (more about addresses later) that can be used in an 8-bit micro. This is because that processor has something called an address bus, and this is 16 bits wide. All this will be explained a little later.

Machine Languages

What, you may be wondering, does all this have to do with my computer, which seems to understand all the Basic programs I feed into it? Well, yes it does understand Basic, but not directly - it has to go through an interpreter. This is rather like being at a United Nations seminar, where people of all nationalities are grouped together and have to communicate with each other. The only problem is, hardly any of them speak the same language. So each of them has an interpreter who converts everything said into the language that the delegate can understand.

The same is true of microprocessors - each one can only understand the language built into it. Although these machine languages come under the generic term of machine code, there is very little similarity between the code for each different processor - unfortunately! This machine code is rather unintelligible to humans, consisting of multitudes of numbers stored in bytes, but to the processor it is the most meaningful thing in its existence!

To translate the Basic program that you type in at the keyboard to the actual code that the processor runs, a second generally larger program (written in the native machine code) interprets each line and calls relevant machine code sections. It stands to reason, though, that this process takes much longer than writing a machine code program in the first place.

Why, then, do high-level languages like Basic exist in the first place? The simple reason is that it is far easier for microcomputer users to write programs in a language nearer to their own than to the machine's language. Before describing these machine codes in detail, it is worth mentioning that even machine code programmers adopt a sort of half way house by writing the programs in something called 'assembly language' (or just 'assembler') rather than actual machine code. This is because machine code is merely a collection of numbers and these are not only very difficult to understand and remember, but they are also horrible to debug - a stage that almost every machine code program has to go through.

Assembly language, on the other hand, is a lot simpler, because it is written specifically for humans. The idea is to give each machine instruction a meaningful mnemonic (this means an aid to memory), so that for instance one would write 'EXX' in preference to 231, although only the latter actually means anything to a Z80 microprocessor. To get from this stage to machine code, a program called an assembler is normally used - this simply takes each mnemonic and converts it into the relevant number.

Hexadecimal

At the same time, it is also interesting that machine code programmers generally prefer to use neither the binary (long strings of zeros and ones) number system or the decimal (base 10, that is, what we humans normally use) system, but instead choose the hexadecimal system. This is base 16, where the digits can represent numbers from 0 to 15. This is more convenient because a number from 0 to 15 can also be represented in four bits, so that any 8-bit (that is, one byte) number can be expressed as not more than two hexadecimal digits. However, the numbers from 10 to 15 in hex (as it is normally abbreviated to) cannot sensibly be written as though they were in base 10, so instead they are represented by the capital letters from A to F. This means that C in hex corresponds to 12, and E represents 14 etc. Thus, any byte-sized number can be written in hex as being between 00 and FF (255), and the convenience of this will soon be seen.

binary	decimal	hex
0000	0	00
0001	1	01
0010	2	02
0011	3	03
0100	4	04
0101	5	05
0110	6	06
0111	7	07
1000	8	08
1001	9	09
1010	10	0A
1011	11	0B
1100	12	0C
1101	13	0D
1110	14	0E
1111	15	0F

Catching Buses

Now to describe the actual processors themselves. We have to take time out from the more tangible aspects of micros for a while so that we can describe the 'internal architecture' of each device. Basically, both processors operate on the same principles, and it is only the finer details of extra facilities offered that differ. As microprocessors look like little black boxes with 40 or so pins, we need to know how they move information around inside the computer. After all, every machine has rather more components than just the processor, and the transferal of information between all these devices is really all a computer does. This brings us onto buses. So, from the outside, each processor has 24 pins of special interest. Eight of these, normally labelled D0 to D7, are called the 'data bus', and the other 16, labelled A0 to A15, are collectively called the 'address bus'. Although the word 'bus' may seem like just another piece of ridiculous jargon, it is in fact almost meaningful in this context, as these buses go from one place to another.

The address bus, as it consists of 16 lines, can represent any number from 0 to 65535 (remember (2 16)-1)), or in hex, from 0000 to FFFF. This bus is used by the processor to tell the memory which particular location it wants information from. So, if the processor wants to get the contents of location C350 (hex), or indeed if it wants to put a new value into this part of memory, then it puts the number C350 on the address bus. A further pin on the processor sends a pulse along to the memory on a different bus (the 'control bus'), and this pulse tells the memory whether it is being read or written (read is known as PEEK and write is known as POKE in Basic). After all this communication (which takes almost no time at all), the memory knows what is happening. If the control bus pulse indicates that data is being read, the memory location being addressed (hence 'address bus') puts its contents onto the data bus (which being 8 bits wide can hold a whole byte's worth of information), and this goes into the processor. Likewise, if the location addressed was told that it was being written to, it would have taken a byte from the data bus and stored it.

This shows that the address bus is unidirectional - that is, information it carries only goes one way, from the processor to the memory, while the data bus is 'bi-

directional', as information can go from the memory to the processor and vice versa. Devices other than the processor can access these buses, but that is beyond the scope of these cards. For further information it is best to consult the relevant device data book.

Registers and Flags

To recap. A Z80 or a 6502 processor can 'talk' to a maximum of 65536 bytes of memory, and this talking can involve both the sending and receiving of information. So far so good, but what can the processor do with this information? Inside a processor is a small area of what is effectively memory, in that not only can it remember things, but it can also be read and written to, just like normal memory. These areas of processor-resident memory are called its registers, and each one is either 1 or 2 bytes long.

At this stage, the differences between the Z80 and 6502 become rather apparent, as the 6502 has only 5 registers, while the Z80 has 22. Quite a difference, but luckily there are common points. Both processors have a (2 byte) register called the Program Counter, a 1 byte register called the Flag or Status register, a 1 byte general purpose register called the Accumulator, and a register called the Stack Pointer, which is 1 byte long on the 6502 and 2 bytes long on the Z80. These are normally abbrieviated to PC, F (Z80) or S (6502), A and SP respectively, and all have particular purposes.

The PC, as its name suggests, keeps track of the program being executed. This program is held in the memory, and the PC holds in memory the address of the next instruction to be executed. Therefore, once the processor has finished carrying out the current instruction, it puts the contents of the PC onto the address bus and so gets the next instruction from the data bus.

The flag register is so-called because it is used to signal particular conditions to both the user and the processor. This is rather like a motorway policeman 'flagging down' a passing motorist to tell him that he is going too fast or that there is a crash up ahead.

The F register (we'll use the Z80 terminology whenever we're referring to the flag/status register) is unique because it is not used in a byte context, but each individual bit is treated as a flag to tell the processor what has recently happened. In the Z80, only 6 of these bits are used, and 7 in the 6502, but only three in each are of immediate interest to us - the Zero flag, the Carry flag and the Sign flag. These are abbreviated to Z, C and S respectively, and are used as follows: If an operation that affects these flags caused a result of zero, the Z flag is set, whilst a non-zero result resets it. Setting means making that bit equal to 1, and resetting means making it equal to 0. Likewise, the carry flag is set if an operation resulted in 'carry' (of which more shortly) and the sign flag indicates whether a result is positive or negative (we haven't discussed negative numbers yet, but don't worry, we will!).

Notice that we said that these flags are changed only if an operation that *affects the flag* in question is used. This is a common stumbling point for novice programmers, as there are so many instructions which can quite easily give results that seem as though they should alter the flags, but in fact they dont't. It may also be rather confusing to think of the Z flag being set to 1 on a zero result, and reset to 0 on a non-zero result. Try to think of it this way — if a zero result was obtained, then the flag is raised, otherwise it is lowered. Make sense?

Now to the operation of the C flag. As we know, a byte can hold any number between 0 and 255, and we can add and subtract two numbers on both processors. So, if we wanted to add 23 and 42, no problem - the answer, 65, can easily fit into a byte. Likewise, subtracting say 11 from 26 gives 15, which again is easily byte-sized. But what happens if we try to add numbers like 200 and 125, or subtract

numbers like 4 from 3? As the answer must fit into one byte, it seems we have a problem, as in both examples, the results 325 and -1 cannot possibly be expressed in one byte. This is where the C flag comes into play - this flag is set if a result is larger or smaller than a byte can represent - exactly similar to the school maths concept of 'carry' in addition and 'borrow' in subtraction. Unfortunately, the designers of the 6502 decided to complicate matters further by making their C flag do the reverse of what you'd expect it to do when subtracting.

Signed Arithmetic

This leads us into the properties of the S flag - the sign flag. We have just discovered that 4 - 3 gives us an answer of -1, but so far we have no real way of representing negative numbers. There is, in fact, no such thing as a negative number as far as processors are concerned, but by following a convention (and note that it is only a convention), we can make mathematicians' lives correspondingly easier. The convention is that if one has chosen to use signed arithmetic, then all the numbers from 0 to 127 (i.e. 00 to 7F in hex) are considered positive, and all the numbers from 128 to 255 (80 to FF hex) are considered to be negative. However, it is FF, not 80, that represents -1, and we count down from here, so that FE is-2, FD is -3, etc., all the way down to 80, which represents -128. This is called 'two's complement' arithmetic, because to arrive at the representation for a particular negative number, you first 'one's complements' the number, which means you turn all the set bits off and turn all the re-set bits on. So, if we wanted to represent -5, we take:

0 0 0 0 0 1 0 1 which is the binary representation of 5. Then we one's complement it thus: 1 1 1 1 1 0 1 0

Now, to turn this into a two's complement number, we simply add 1 to it. Therefore, we are left with: 1 1 1 1 1 0 1 1

This number, which also happens to represent 251 in normal unsigned arithmetic, is the conventional representation of -5 in an 8-bit machine. So, where does the S flag come into it? First of all, this flag only means what it is supposed to mean if you *are* following this convention. As we know that all positive numbers are from 00 to 7F, and that all negative numbers are from 80 to FF, it follows that we can tell if a byte is negative very simply - the seventh bit is set. This is because the seventh bit respresents 2^7, which is 128, and all numbers below 128 are positive. Therefore this bit is never set for a positive number, but is always set for negative numbers. So, the S flag simply reflects the state of this seventh bit - it is set if the number in the accumulator is over 7F; that is, if it is negative.

The point of having all these flags is that it is possible to make the program's execution point jump from one location to another (rather like Basic's GOTO statement), and these jumps can be prompted to occur on certain conditions, the presence or absence of these conditions being indicated by the state of the flags. In this way, it is possible to jump if the Z flag is set, for example, and so programming loops similar to FOR . . NEXT loops can easily be implemented.

Now we move onto the stack pointer register (SP), and again the differences between the Z80 and 6502 become apparent. The Z80's SP is 16 bits (i.e. 2 bytes) long which means that 'the stack' can be placed anywhere in memory, whilst the 6502's SP is only 8 bits long. This is because the 6502's stack is *always* in memory locations 100 to 1FF (hex), and so its SP only needs to point to the actual byte within this 'page', where a page is a block of memory 256 bytes long. Zero page (which is very important to the 6502) extends from address 00 to FF, page three extends from 300 to 3FF, and so on. Therefore, the stack is always in page 1 on a 6502. This restriction means that not only must this area of memory always be set asisde for the stack, but the stack can never be bigger than a page. For most of

us, this is not important, but certain complex control applications use up a lot of stack space, and this can pose problems.

The Stack

What, then, is the stack? Well, it is an area of memory used by the processor and the user as a temporary store of virtually anything at all. The machine uses it for storing return addresses (see later), while the user may wish to save the current values of various registers for a while, which all makes the stack a very convenient place. To know exactly where to put things onto this stack, the processor uses the SP, which points to the next free place on the stack. When it has to put something new on the stack, it places it where the SP is pointing, and then updates the SP to point to the next free place after this. Likewise, when it removes something, it takes from where the SP is pointing and changes the SP to point to the location just prior to where it was before.

This system is described as a 'Last In First Out' (LIFO) stack. Perhaps the best way of visualising this stack is to use the time-worn analogy with the spike used as an impromptu filing system in offices worldwide. With these little devices, office staff place a sheet of paper to be filed on the spike, on top of everything else that is already impaled there, so that this last sheet placed is the first to be removed. To get at an item lower down, the clerk has to remove all the sheets above it. This is exactly how a microprocessor stack works, except that everything is upside down - equivalent to sticking the spike to the ceiling. As an aside, it is actually possible to manipulate the SP and reach items further down the stack, but this is really for rather more advanced programmers, and is best left alone by newcomers to programming. The process of putting items onto the stack is normally called 'pushing' (the Z80 mnemonic for this operation is PUSH), and the removal process is called 'popping' (Z80) or 'pulling' (6502).

The other item common to both the Z80 and the 6502 is the accumulator or A register. As we've already discovered, this is an 8-bit general purpose register, and it is the operations on this register more than any other that affect the flags. It is called the accumulator more for historical reasons than any other, especially with regard to the Z80, where all the registers share a similar versatility. In the first few processors designed, it was only the accumulator that could be used as the calculation and manipulation register, and this traditional name seems to have stuck despite the great leaps in technology since then. It is still true, however, that more can be done to the A register than to any other, and it can be added to, subtracted from, rotated and shifted, logically operated on (using OR, AND and XOR), incremented and decremented (which means increasing or decreasing its value by 1), loaded from memory, stored in memory and compared with other values. This list covers almost all the operations possible in machine code - these are all you need to do absolutely any computing application - honest!

Here are some other common operations. First of all, shifting. As we know that a byte is composed of 8 bits and that each bit represents a power of 2, a quick resort to simple maths shows that if we could move all the bits along one place to the left, the result would be two times the original value. Likewise, if we shifted everything one bit to the right, the result would be half of the original. Such operations can indeed be done, and they are called, surprisingly enough, shifts. There are, of course, problems with this - shifting a byte left that has a '1' in bit 7 will mean that this bit is lost, and likewise shifting a set bit 0 right will lose that bit too. So, to get over this, the designers have very clearly made sure that the bit 'falling off' will be transferred to the carry flag.

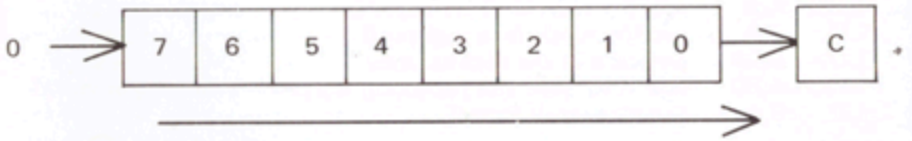
But what if the number was a signed number following our earlier convention of having the seventh bit set if negative? Shifting such a number right would obviously

corrupt the sign and produce meaningless results, so there is a special instruction, called 'Arithmetic Shift Right' to handle this, which ensures the seventh bit is copied back into itself in order to preserve the sign.

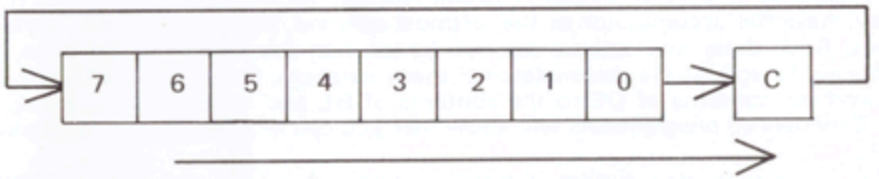
In the same instruction type is the Rotate instruction. This is basically the same as a shift, except that instead of putting a 0 into the empty bit (see below), the content of the opposite end of the byte is rotated to the other end. (The diagram below shows this rather more clearly).

Logical shift left

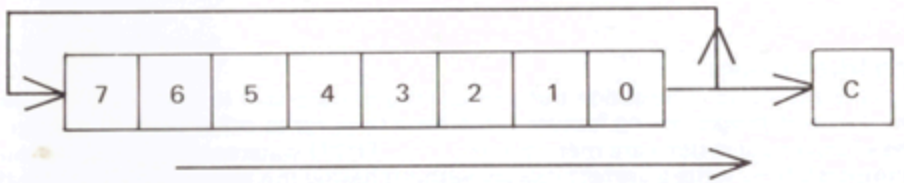
Logical shift right



Rotate right carry



Rotate right



There is also a further variant on the Z80, where the byte is made effectively one bit longer by considering the carry flag as the ninth bit. This is useful in multiplications involving numbers larger than one byte, where the result of a shift or rotate needs to be carried across to the next byte.

We then have the logical operations; OR, AND and XOR (EOR in 6502 - both stand for 'Exclusive or'). These operators were described in detail in the November issue of *Which Micro?* in an article called 'Spectrum Graphics', but for those of you that missed it, here is a brief synopsis. All these function are described as operating 'bitwise' in machine code, as they treat each operand as a collection of eight individual bits rather than a byte. OR will set the relevant bit in the result if either or both of the operands have this bit set, AND will set the result bit only if both operand bits are set, and XOR works just like OR except that if both operand bits are set then the result bit is reset. All are extremely useful in a surprising number of applications, such as ensuring that registers contain zero, masking off unrequired bits of a data byte and toggling bits.

Subroutines

A major feature that needs explanation is the concept of call and return. Those familiar with Basic will know all about subroutines and how they are called using the Basic keyword GOSUB. A subroutine (under any language system) is a part of the program likely to be used more than once. For reasons of memory efficiency it is useful to have just one version of this part of the program, that can be 'called' by all the other parts of the program when they need it. In the few applications where speed is critical it is occasionally the case that subroutines are not used. However, with machine code being very much faster than any version of Basic, the marginal increases in speed (on a Spectrum avoiding subroutine calls will save you about a microsecond) are not worth the extra memory involved. The same sort of thing happens in machine code too, with the relevant instructions here being CALL in Z80 and JSR in 6502 (JSR stands for 'jump to subroutine' - a not strictly accurate mnemonic). When the processor encounters one of these instructions it saves the address of the NEXT instruction onto the stack, adjusting the stack pointer as it does so. This address is called the return address. It then loads the two bytes following the CALL instruction into the PC, treating them as an address. This is the call address, and it is the programmer's responsibility to ensure that this is a valid address.

Now that the call address is in the PC, the processor continues execution from this point until it reaches a RET (RTS in 6502) instruction. This retrieves the return address from the stack, adjusts the stack pointer and then proceeds to load the return address into the PC, so that the processor will carry on from where it left off - just after the original CALL instruction. As the call and return mechanism is so dependent on the stack, it is very important that this is not corrupted (programmers tend to say 'unbalanced', but then so are many programmers!) during a subroutine. In other words, if you put anything on the stack, you must take it back off again before you execute a RET. Interestingly, experienced programmers tend to use this tack to deliberately corrupt the stack and effect a sort of 'calculated return' to a different address than from whence it came. As subroutines are of fundamental importance to any machine code program, spend some time on this section and get it clear in your own mind.

The Z80

This machine, apart from the common registers mentioned above, also has a whole host of others: B, C, D, E, H, L, A', B', C', D', E', F', H', L', IX, IY, I and R. This dazzling array of registers is unparalled on any other 8-bit processor, and programming this device is a much simpler task than programming the 6502. The clever thing about the Z80 registers is that they can be joined into groups of two, the new 'register pair' being perfectly useable as a 16-bit register. Thus, B and C

continued overleaf →

become BC, D and E become DE and H and L become HL. This increases versatility to a great degree, as not only can these 16-bit registers be used as such, including arithmetic operations such as add and subtract, increment and decrement, but they can also still be used as the individual registers comprising each pair. Note that only the register pairs described above can be used, in other words DL or BE cannot exist. These register pairs can be easily manipulated, and can be used for stack operations, address manipulation, indirection vectors (i.e. pointing to memory for the purposes of other registers) and general 16-bit calculations.

The 'alternative' register set, A' to L', is identical in every respect to the other set, but only one set can be in use at at time. This alternate set is very valuable in that the Z80 instruction to swap sets is so fast that it is often used as an alternative to saving all the registers on the stack. Of the four registers left, only IX and IY are important to us, as I and R are not really meant for programming *per se*.

IX and IY are known as the index registers and are both 16 bits long. They both operate in exactly the same way - they are normally loaded with the base address of some data or similar, and then used as an 'index' together with an 'offset' specified in the machine code, so that the user can access any element of this table that he requires. With versatility like this, it is sometimes hard to understand why so many manufacturers still use the 6502!

The Z80 instruction set (in other words, all the possible instructions and operations) is also vast, as anyone who has seen the back of a Spectrum or ZX81 manual will know! We will take the main instruction mnemonics in turn and explain what each does.

The most important of these is LD, which stands for 'LoaD', and is the Z80's do everything instruction. LD always has two operands, the destination and the source, and both of these can be either registers or memory. Note that the source register/memory location is not affected by the LD instruction. For some totally unknown reason, the Z80 allows the destination and the source registers to be the same - obviously pointless instructions! As examples, here are a few:

LD A,B - put the contents of register B into register A.

LD C,(HL) - put the contents of the memory location addressed by the contents of the register pair HL into register C. This is like saying 'LET C = PEEK HL' in Basic.

LD A,(12345) - put the contents of memory location 12345 into register A.

LD HL,(23456) - put the contents of memory locations 23456 and 23457 into the HL register pair. (Two bytes are transferred because HL is 2 bytes long.)

LD (DE),A - put the contents of register A into the memory location pointed to by the contents of DE. This is equivalent to POKE DE,A' in Basic.

LD (34567),BC - put the contents of the BC register pair into memory locations 34567 and 34568.

LD instructions do not change any of the flags. As can be seen, LD is very versatile and does a lot of very useful things. By itself, however, it can't do much, we need more instructions: DEC and INC. DEC stands for 'decrement' and means reduce by 1, so

DEC C - means take 1 from the contents of register C.

DEC HL - reduce the contents of HL by 1.

Note that if the decrementing takes a register below 0, it becomes FF(255). A register pair becomes FFFF (65535). Conversely, INC stands for 'increment' and this means increase the value by 1. If this makes a register cross FF it becomes 0, and if a register pair crosses FFFF it also becomes 0. INC and DEC on single registers or on indirected memory (as in DEC (HL)) affect the zero flag, so that it becomes set if the result becomes 0, otherwise it is reset, while the INC and DEC on register pairs do not affect this flag. Neither of them change the carry flag.

Addition and Subtraction

Now that we can move data between memory and between registers, and alter the value of this data by a value of 1 either way, the next step is to introduce adding and subtracting. There are two different types of addition on the Z80 - straight addition and addition including the current state of the carry flag. The mnemonics for these are ADD and ADC respectively. ADD simply takes two operands and adds them together, changing the carry and zero flags as necessary but not taking them into account during the actual addition. Therefore, if we wanted to add 234 and 73 and the carry flag was set, after an addition using ADD the result would be 51 (as 234 + 73 is 256 + 51) and the carry flag would still be set, as the result went over 255.

On the other hand, using ADC in exactly the same circumstances would return a result of 52, as the carry flag indicates that 1 should be added into the addition as a carry over from the last operation. This is very useful when adding together numbers that result in the answer being greater than 255, and just as an example, here is a short routine to do just that: add 234 and 73 together *and* return the right answer:

```
LD   A,234   put 234 into the accumulator
LD   B,73    put 73 into register B
AND  A       this clears the carry flag (resets it) without affecting A
ADD  A,B     add the two numbers together
LD   C,A     put the result into register C
LD   A,00    put zero in the accumulator
ADC  A,00    add A to itself but including the carry (i.e. 0 + 0 + carry)
LD   B,A     put this result into B
```

The result is now in the register pair BC, and it will be found that C contains 51 and B contains 1. So, treating this as 16 bits, the actual result returned is 256 + 51 = 307, which is correct.

However, that routine is a little long-winded and would in practice be cut down fairly considerably in an actual program. Notice that both ADD and ADC must always have the accumulator as the leftmost operand, and this is where the result is put. Both these instructions can also be used on the 16-bit register pairs, HL taking on the role of the accumulator in these instances. For example ADD HL,DE will add the contents of DE to the contents of HL and leave the same answer in HL. Experienced programmers will know that you can also use IX and IY in place of HL.

Subtraction is very similar, the mnemonics being SUB and SBC, the latter taking the carry flag into account before the subtraction. Again A is always one of the registers and will always hold the result. Likewise, 16-bit register pairs can also be subtracted using HL (or IX/IY), but this time there is no 16-bit SUB - only SBC can be used.

Condition Codes

Very similar to subtraction but perhaps more important in practice is comparison. A major programming feature is the need to jump to other parts of a program when certain conditions are met - the IF. . . . THEN statement in Basic. It would be impossible to write a useful program without having this ability. This is where the importance of the flag register comes into play, along with the ability to compare things. This can only be done one byte at a time (in other words there is no register pair equivalent) and the leftmost operand is always the A register. Therefore it is always left out of the mnemonic (just like AND, OR and XOR). So, CP B means 'compare the contents of register B with the accumulator' and CP 26 means

'compare the number 26 with the accumulator'. Compare, here means performing a SUB changing the flags as it does so, but not actually putting the result anywhere. Thus, the accumulator is not altered after a comparision, but the flags are. So, if A is bigger than the operand, neither the zero nor the carry flag are set. If the A register is the same as the operand, the zero flag is set and the carry flag is set. Action can then be taken on the result of the comparison.

This brings us neatly round to condition codes. These are parts of mnemonics that tell the processor to follow the instruction only if the flags are in the requisite state. The most common ones are Z, NZ, C and NC. These stand for 'if zero set', 'if zero reset', 'if carry set' and 'if carry reset' respectively. These condition codes can be applied to four types of instruction:- RET, CALL, JP and JR.

We have already discussed CALL and RET, but what about JP and JR? JP stands for 'jump' and JR stands for 'jump relative'. Jumping is identical to Basic's GOTO, and simply passes control to another point in the program. JP is therefore followed by an address, and control will pass to this address if the condition after JP is satisfied. JR is the same except that it is followed by only one byte (an address is always two bytes long), and this byte is taken as an offset to the program counter rather than its new value. It also follows the two's complement convention, so that it is taken as a positive offset if it is between 0 and 127, but as a negative offset if it lies between 128 and 255.

Note that both JP and JR, just like CALL and RET, do not have to be followed by a condition code, so that the jump is taken no matter what the state of the flags. Here is a short example:

```
LOOP:   LD   A,(HL)   load A with the contents of the memory location pointed
                      to by HL
        CP   23       compare this value with the number 23
        JR   Z,OUT    if they are the same, jump to 'OUT'
        INC  HL       increase HL by 1
        JR   LOOP     jump back to 'LOOP' unconditionally
OUT:                  rest of program
```

Notice here the use of 'labels' to make assembly language more readable. The labels here actually represent addresses, and all of these are calculated by the program that turns the assembly language into machine code (called an 'Assembler'). The short routine above looks in memory for the value 23, and when successful it jumps to 'OUT' and continues with the rest of the program.

The Z80 programmer is very fortunate, as the instruction set also includes one very valuable sort of relative jump - the DJNZ instruction. This is like a FOR . . . NEXT loop in Basic, with the B register being used as the loop counter. This is what happens.

```
        LD   B,34     this is the loop counter - the number can be from 0 to 255
LOOP:                 from here down to the DJNZ instruction is a routine that does
                      whatever is required but must not affect the B register - unless
                      this is particularly desired. It must be short enough to fit within
                      the jump range of a relative jump.
                      DJNZ LOOP
                      rest of program
```

Here, the B register is set up as a counter, and the section of code between LOOP and the DJNZ instruction is executed B times. This is because DJNZ means 'decrement and jump (relative) if non-zero'. This means decrement B each time round the loop, and if it has not reached zero yet, jump back to the label 'LOOP'. This is an extremely useful instruction.

The Z80 also has a fully comprehensive set of shift and rotate instructions, the

actions of which were described earlier.

Another set of instructions unique to the Z80 is the BIT, SET and RESET groups of instructions. They are all concerned with individual bits within a byte. SET and RESET do just that, setting or resetting the requisite bit as required. BIT is a testing instruction that changes the zero flag according to the value of the bit being tested. Thus, BIT 3,C will set the zero flag if bit 3 of C is reset, and will reset the zero flag if bit 3 of C is set. Although all these actions can be simulated using the logical operators (OR, AND and XOR) with suitable 'masks', having the instructions there at your fingertips takes a lot of the calculating out of programming.

Further Z80 Facilities

Having discussed the major instructions on the Z80, we are left with a whole host of miscellaneous operations. So here's a quick description of each. CPL operates on the accumulator and simply inverts all the bits in it. (CPL stands for 'complement' and is identical to one's complementing the accumulator). NEG does just the same but adds 1 to the result - in fact, this is two's complementing it. SCF sets the carry flag, while CCF complements (inverts) it. There is no clear carry flag instruction, but AND A does this without affecting any registers. It also clears the zero flag. NOP does absolutely nothing but waste time, HALT stops the processor until an interrupt is received, EI and DI enable and disable the interrupts respectively, and IM 0, IM 1 and IM 2 select interrupt modes as appropriate. This is rather advanced stuff and will not normally be used by programmers.

EX DE,HL swap the contents of HL and DE; EC AF,AF' swap between the accumulator and flag registers in the alternate set; EXX does the same for B, C, D, E, H and L. EX (SP),HL swaps the top of the stack with the contents of HL. This leaves us with the restart (RST) instruction. This is simply a CALL instruction occupying 1 byte, and can be useful - although only 8 addresses can actually be addressed, it is much faster and much shorter (in terms of memory space) than an equivalent CALL instruction.

Before any knowledgeable readers scream and shout, yes there are a lot more instructions and techniques we haven't covered, but these are unlikely to be of great use to the novice machine code programmer.

Addressing Modes

We are now on the verge of delving into the 6502 processor, but before we do we must talk a little bit about addressing modes. We have in fact used many different addressing modes in our discussion of the Z80, but we didn't tell you this!

If we write 'LD A,58' then we are using what is known as immediate addressing - the operand is specified in the mnemonic and thus the resultant machine code. Writing 'RET' or 'POP HL' uses implied addressing, as the operand is either mentioned in the mnemonic itself or else there is no operand. LD A,B is an example of absolute addressing where both operands are specified in the mnemonic. Possibly the hardest Z80 addressing mode to understand is indirect addressing: LD A,(HL) and LD DE, (12345) are both examples of it. Here, the first means to load the accumulator with the contents of memory location pointed to by the contents of HL (in other words, if HL contained 54321 then load A from this address) and the second means load DE with the contents of memory location 12345 (into E) and 12346 (into D). Another form of this mode is indexed indirect, where one of the index registers, IX and IY, are used. Thus LD A,(IY+3) means to load the accumulator with the contents of the memory location pointed to by IY+3. In other words, if IY contained 24576 then this mnemonic means load A from

address 24579. If these are confusing, pity the poor 6502 programmers; for them the methods discussed so far are just a small selection from a morass of very confusing addressing modes!

The 6502

Let's discuss that very processor a little more fully. Unlike the Z80 it only has two other registers: X and Y. These are both described as index registers, but being only 8 bits long, their versatility is somewhat less than their Z80 equivalents. The flags on the 6502 are affected by almost every instruction, even the 6502's equivalent of LD - which is LDregister. Thus, LDA means 'load the accumulator', LDX means 'load register X', and LDY means 'load register Y'. However, LD on the 6502 is not quite the same as LD on the Z80, as it does not invoke register-register loads (which are done by TX - transfer - instructions) or register-memory loads (which are handled by ST - store - instructions). Therefore, we can have:

LDA 23	which means put the number 23 in A; immediate addressing
LDA 23	which means put the contents of address 23 in A; zero page absolute (see below for zero page details)
LDA 2345	which means put the contents of address 2345 in A; absolute
LDA 2345,X	put the contents of address (2345+X) in A; absolute indexed X
LDA 2345,Y	put the contents of address (2345+Y) in A; absolute indexed Y
LDA (12),Y	means take the contents of memory address 12 along with the contents of memory location 13, treat this as a 16-bit pointer to another memory location, add Y to this address and load the accumulator with the contents of this address. Note that the left hand operand must be in zero page. This is known as 'indirect indexed addressing'.
LDA (23,X)	If the contents of memory location 23 and 24 together make up an address that points to a table of pointers (get the point!) then this means load the accumulator with the contents pointed to by the Xth element in this table. As above, the lefthand operand must be in zero page.

As you can see, not only are the 6502's addressing modes extremely complicated, but this concept of Zero Page is fundamental. We have already described exactly what zero page is, but it must always be remembered that this area of memory is one of the most important to the processor. The reason for specifying a page of memory like this is that operations using zero page are significantly faster than other page equivalents. One reason for this is because the top 8 bits of a zero page location will always be 0, they can be left out of the mnemonic and resultant machine code, thus making the actual instruction one byte shorter.

There are also LDX and LDY instructions in 6502, but obviously they are not quite as versatile as the LDA group. Notice the use of the '#' hash sign in 6502 assembly language to signify immediate addressing.

STA, STX and STY are very similar to LDA etc, but do roughly the opposite — rather than loading a register from memory, the store instructions load memory from a register.

The final part of the LoaD, STore group are the transfer instructions. These are very simple, as they all use implied addressing, thus:

TAX copy the contents of A into X
TAY copy the contents of A into Y
TSX copy the stack pointer contents into X
TXA copy the contents of X into A
TXS copy the contents of X into the stack pointer
TYA copy the contents of Y into A

We now have all the 6502's equivalents of the Z80's LD instructions. Because of the sheer range of addressing modes on the 6502, some instructions seem more powerful than the corresponding Z80 ones. For instance, the logical operators (OR, AND and EOR) have the same range of addressing modes as LDA, which means that the accumulator can be logically operated on by almost anything. Like wise, INC and DEC can change memory directly. Note however that although X and Y can be incremented and decremented (INY, INX, DEY, DEX) the accumulator cannot.

Perhaps the most powerful instruction on the 6502 compared to its Z80 equivalent is the compare instruction. Not only are all the LDA addressing modes available to it, but there are also CPY and CPX instructions, which allow the programmer to compare something with the contents of X or Y. On the Z80, the only register which can be used with the compare instruction is the accumulator.

6502 Limitations

However, the more specialised instructions, such as shifting and rotating, are much less comprehensive. There is only one of each shift and one of each rotate instruction available. There are ASL and LSR (shift left and shift right respectively) and ROL and ROR (rotate left and rotate right). The jumping instructions (which are JP and JR on the Z80) are called branching instructions on the 6502, and all except JMP itself are relative jumps. JMP can have no condition codes attached, and in fact conditional branches apart from JMP are all that exist. Nevertheless, many people have placed a great deal of faith in this processor, and rivalry between the 6502 and the Z80 will continue until 8-bit processors are totally forgotten.

Finally, to demonstrate assembly language on both processors, we will show a program that adds two 16-bit numbers together to leave a 24-bit result. In order to get at this result from Basic, you'll have to treat each of the four bytes that comprise the result as being 256 times more significant than the one before it. Thus, if the result was placed in memory locations FFFO upwards (65520 decimal), then the result could be obtained like this; LET result = PEEK (65520) + 256* PEEK (65521) + 65536* PEEK (65522)

LD HL, (NUM1)	put the two numbers being added together (assuming that
LD DE, (NUM2)	they had been stored in memory locations NUM1 and NUM2) into the HL and DE register pairs.
AND A	clear the carry flag
ADD HL, DE	add the two numbers together — any overflow over 65535 will set the carry flag
LD (RES), HL	put the lower 16 bits into the first two result memory locations
LD A, OO	put zero in A
ADC A,A	add A to itself. This will simply put the state of the carry flag into A as A itself contains zero. It could also have been done with an RLA instruction.

```
LD (RES+2), A   put the top 8 bits of the result into the last result memory
                location.
RET             return to the calling routine.
```

As you can see, it is a very simple process in Z80 assembly language! Now for the 6502 version:-

```
LDA NUM1        get the lowest 8 bits of the first number into A
CLC             clear the carry flag
ADC NUM2        add the lowest 8 bits of the second number into A
STA RES         store the lowest 8 bits of the result
LDA NUM1+1      get the top 8 bits of the first number in A
ADC NUM2+1      add in the top 8 bits of the second number including the
                carry (if any) from the last additon
STA RES+1       store the middle 8 bits of the result in the correct place
LDA # 0         this operation and the one below do just what the Z80
ADC # 0         program did; take the carry from the previous addition and
                put it into A
STA RES+2       store the top 8 bits of the result
RTS             return to the calling routine.
```

As you can see machine code reallys *is* simple!!

MACHINE CODE

Made Easy

FREE